A Parallel Algorithm for Compile-Time Scheduling of Parallel Programs on Multiprocessors

Yu-Kwong Kwok and Ishfaq Ahmad

Email: {csricky, iahmad}@cs.ust.hk

Department of Computer Science
The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong.

Abstract†

In this paper, we propose a parallel randomized algorithm, called *Parallel Fast Assignment using Search Technique* (PFAST), for scheduling parallel programs represented by directed acyclic graphs (DAGs) during compile-time. The PFAST algorithm has $O(e)$ time complexity where e is the number of edges in the DAG. This linear-time algorithm works by first generating an initial solution and then refining it using a parallel random search. Using a prototype computer-aided parallelization and scheduling tool called CASCH, the algorithm is found to outperform numerous previous algorithms while taking dramatically smaller execution times. The distinctive feature of this research is that, instead of simulations, our proposed algorithm is evaluated and compared with other algorithms using the CASCH tool with real applications running on the Intel Paragon. The PFAST algorithm is also evaluated with randomly generated DAGs for which optimal schedules are known. The algorithm generated optimal solutions for a majority of the test cases and close-to-optimal solutions for the others. The proposed algorithm is the fastest scheduling algorithm known to us and is an attractive choice for scheduling under running time constraints.

Keywords: Compile-Time Scheduling, Task Graphs, Multiprocessors, Parallel Processing, Parallel Programming Tool, Parallel Algorithm, Random Search.

1 Introduction

To efficiently exploit the tremendous potential of parallel architectures, the tasks of a parallel program must be carefully decomposed and scheduled to the processors so that the program completion time is minimized. When the characteristics of the parallel program, such as execution times of the tasks, amount of communication data, and task dependencies are known *a priori*, the parallel program can be modeled as a node- and edge-weighted *directed acyclic graph* (DAG) $G = (V, E)$, in which V is the set of nodes‡ and E the set of edges, representing tasks and messages, respectively. The weight associated with a node represents the amount of execution time of the corresponding task and the weight associated with an edge represents the amount of communication time. An example DAG is shown in Figure 1 (n_i 's are the indices of nodes). With such a *static* model, the scheduler is invoked off-line during compile-time and thus can afford moderate time complexity in order to generate a better schedule. This form of multiprocessor scheduling problem is called *static scheduling* or DAG scheduling [1], [4], [6], [8], [15]. Static scheduling, even with a very simple model, is an NP-complete problem [5], [7]. For instance, the problem is NP-complete even in two models: (1) scheduling unit-weighted tasks to an arbitrary number of processors [7], (2) scheduling one or two unit-weighted tasks to two processors [5]. Optimal solutions exist only in three simple cases: (i) scheduling a tree-structured DAG with identical node weights to an arbitrary number of processors [5], (ii) scheduling an arbitrary DAG with identical node weights to two processors [5], and (iii) scheduling an interval-ordered DAG to an arbitrary number of processors [5]. However, even in these cases, no communication is assumed among the tasks of the parallel program. Thus, heuristic approaches are sought to tackle the problem under more realistic cases in a reasonable amount of time [8], [11], [15], [16].

While it is understood that static scheduling is done *off-line* and therefore some extra time can be afforded in generating a better solution, the time-complexity of a scheduling algorithm is an important issue from a practical point of view. Although there are a large number of scheduling heuristics suggested in the literature and many of them can generate good solutions, few have a low time-complexity [6], [10], [13], [14], [18]. As such most of the algorithms may be inapplicable for practical purposes. In a recent study [2], we compared 21 recently reported algorithms and made a number of observations. For example, we found that an $O(v^3)$ algorithm may take more than an hour to produce a schedule for a DAG with 1,000

†. This research was supported by the Hong Kong Research Grants Council under contract number HKUST179/93E.

‡. Throughout the paper we denote $|V|$ and $|E|$ by v and e, respectively.

Proceedings of the 1997 Conference on Parallel Architectures and Compilation Techniques (PACT '97)
0-8186-8090-3/97 $10.00 © 1997 IEEE

nodes, a typical size for many applications [2], [20]. Taking such a large amount of time to generate a solution for an application is a major hurdle in incorporating these algorithms in parallelizing compilers. On the other hand, some algorithms have low time-complexity but their solution quality is not satisfactory [2]. Thus, an algorithm which meets the conflicting goals of high performance and low time-complexity is highly desired. In this regard, Yang and Gerasoulis [21] proposed some novel techniques for reducing the time-complexity of scheduling algorithms. Our objective is to design an algorithm that has a comparable or lower complexity while producing better solutions.

In this paper, we propose a low complexity scheduling algorithm called *Parallel Fast Assignment using Search Technique* (PFAST) which has $O(e + v)$ time-complexity and is a parallel algorithm. The PFAST algorithm is based on an effective search technique. The linear-time algorithm first generates an initial solution and then refines it using a random neighborhood search technique. In addition to simulation studies, the algorithm is evaluated using a prototype computer-aided parallelization and scheduling tool called CASCH (Computer-Aided SCHeduling) [3], with real applications running on the Intel Paragon. The PFAST algorithm outperforms numerous previous algorithms while its execution times are dramatically smaller. Indeed, based on our experimental results, the PFAST is the fastest scheduling algorithm known to us. The algorithm is also evaluated using random task graphs for which optimal solutions are known. The PFAST algorithm generated optimal solutions for a significant portion of the test cases and close-to-optimal solutions for the other cases. Furthermore, the algorithm is scalable in that it exhibits an almost linear speedup. The PFAST algorithm is therefore an attractive choice for generating high quality schedules in a parallel processing environment under running time constraints.

This paper is organized as follows. In Section 2, we first discuss the trade-off between more complexity and better performance. In the same section we introduce the idea of using neighborhood search to improve the scheduling performance. In Section 3 we describe the proposed PFAST algorithm and its design principles. In Section 4 we present a detailed example to illustrate the functionality of the algorithms. Section 5 contains the performance results of the PFAST algorithms as well as a comparison with other algorithms. The final section concludes this paper.

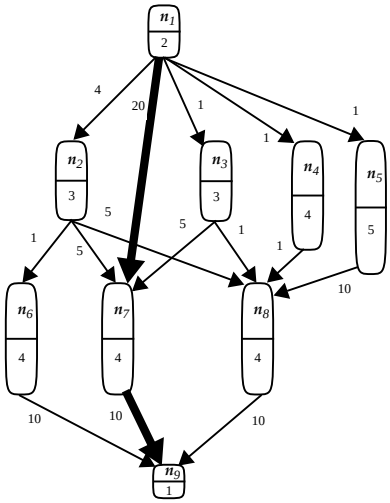


Figure 1: A simple DAG.

2 Related Work and Motivation of a New Approach

Traditional DAG scheduling algorithms attempt to minimize the schedule length through local optimizations of the scheduling of individual nodes. However, most of the local optimization strategies are not effective in general in that most algorithms minimize the start-time of a node at each scheduling step. These algorithms differ only in the way of selecting a node for scheduling. Some of them construct a list of nodes before scheduling starts (as in the list scheduling algorithms like the MCP algorithm [20]) while some of them dynamically select nodes for scheduling (e.g., the DLS algorithm [19]). However, in an optimal schedule, some nodes may have to start later than the earliest possible time. Thus, like most algorithms of a greedy nature, these scheduling algorithms cannot avoid making a local decision which may lead to a sub-optimal schedule. As backtracking is not employed in order not to incur high complexity, a mistake made in an earlier step may not be remedied in later steps.

To obtain an optimal schedule, we have to tackle the scheduling problem from a global perspective. However, global information is usually obtained at the expense of high time-complexity. To obtain such global information, we can use the characteristics of the task graph such as the graph structure and the relative magnitudes of the node and edge weights. Using such attributes, we can decide, from a global point of view, which nodes in the task graph deserve special attention so that eventually an optimal schedule can be constructed. For instance, some previously reported scheduling algorithms is based on a global characteristic structure of the task graph, namely, the *critical path* (CP). A CP is a path with the maximum sum of node and edge

Proceedings of the 1997 Conference on Parallel Architectures and Compilation Techniques (PACT '97)
0-8186-8090-3/97 $10.00 © 1997 IEEE

weights or simply, the maximum *length*. Thus, if an unlimited number of processors are available, the length of the CP provides an upper bound on the schedule length. In light of this attractive property, most scheduling algorithms assign higher priorities to nodes of the CP for scheduling. For example, in most list scheduling algorithms, a node on the CP occupies an earlier position in the scheduling list. However, while the CP length provides an upper bound on the schedule length, making all the nodes on the CP start at the earliest possible time does not guarantee an optimal schedule. In fact, if the edge weights are much larger than the node weights in general, such a strategy can even lead to a bad schedule [2].

To meet the conflicting goals of high performance and high efficiency, we employ an effective optimization technique—neighborhood search [17]. In simple terms, in a neighborhood search algorithm, an initial solution with moderate quality is quickly generated. Then, according to some pre-defined neighborhood, the algorithm probabilistically selects and tests whether a near-by solution in the search space is better or not. If it is better, adopt it and start searching in the new neighborhood; otherwise, select another solution point. Usually the algorithm stops after a specified number of search steps has elapsed or the solution does not improve after a fixed number of steps. The success of such neighborhood search techniques chiefly relies on the construction of the solution neighborhood. A judiciously constructed neighborhood can potentially lead the search to attain the global optimal solution.

3 The Proposed Algorithm

In this section, we present the proposed PFAST algorithm and its design principles. To facilitate understanding of the neighborhood search technique, we first restrict the discussion to the sequential version of the PFAST algorithm, which is referred to as simply the FAST algorithm. We then describe the parallelization technique leading to the PFAST algorithm. A detailed scheduling example will be presented in Section 4.

3.1 A Solution Neighborhood Formulation

Neighborhood search is an old but effective optimization technique. The principle of neighborhood search is to refine a given initial solution point in the solution space by searching through the neighborhood of the initial solution point. To apply the neighborhood search technique to the DAG scheduling problem, we have to define a neighborhood of the initial solution point (i.e., the initial schedule). We can arrive at such a neighborhood definition by using the observation discussed below.

A simple neighborhood point of a schedule in the solution space can be defined as another schedule which is obtained by transferring a node from a processor to another processor. In the DAG scheduling problem, one method of improving the schedule length is to transfer a *blocking-node* from one processor to another. The notion of *blocking* is simple—a node is called blocking if removing it from its original processor can make the succeeding nodes start earlier. In particular, we are interested in transferring the nodes that block the CPNs (CP nodes) because the CPNs are the more important nodes. However, high complexity will result if we attempt to locate the actual blocking-nodes on all the processors. Thus, in our approach, we only generate a list of *potential* blocking-nodes which are the nodes that may block the CPNs. Again, to maintain a low complexity, the blocking-nodes list is static and is constructed before the search process starts. A natural choice of blocking-nodes list is the set of IBNs and OBNs† (with respect to an initial CP) because these nodes have the potential to block the CPNs in the processors. In the schedule refinement phase, the blocking-nodes list defines the neighborhood that the random search process will explore. The size of such a neighborhood is $O(vp)$ because there are $O(v)$ blocking-nodes and p processors.

3.2 Scheduling Serially

To generate an initial schedule, we employ the traditional list scheduling approach—construct a list and schedule the nodes on the list one by one to the processors. The list is constructed by ordering the nodes according to the node priorities. The list is static so that the order of nodes on the list will not change during the scheduling process. The reason is that as the objective of our algorithm is to produce a good schedule in $O(e + v)$ time, we do not re-compute the node priorities after each scheduling step while generating the initial schedule. Certainly, if the schedule length of the initial schedule is optimized, the subsequent random search process can start at a better solution point and thereby generate a better final schedule.

In the FAST algorithm, we use the CPN-Dominant List as the scheduling list. The CPN-Dominant List can be constructed in $O(e + v)$ time since each edge is visited only once.

Construction of the CPN-Dominant list:

(1) Initially, the list is empty. Make the entry CPN be the first node in the list. Set *Position* to 2.

†. An IBN (In-Branch Node) is an ancestor of a CPN but is not a CPN in itself. An OBN (Out-Branch Node) is a node which is neither a CPN nor an IBN.

Let n_x be the next CPN.

Repeat

(2) **If** n_x has all its parent nodes in the list **then**

(3) Put n_x at *Position* in the list and increment *Position*.

(4) else

(5) Let n_y be the parent node of n_x which is not in the sequence and has the largest *b-level*†. Ties are broken by choosing the parent with a smaller *t-level*. If n_y has all its parent nodes in the sequence, put n_y at *Position* in the sequence and increment *Position*. Otherwise, recursively include all the ancestor nodes of n_y in the sequence so that the nodes with a larger value of *b-level* are considered first.

(6) Repeat the above step until all the parent nodes of n_x are in the list. Then put n_x at *Position* in the list.

(7) **endif**

(8) Make n_x to be the next CPN.

Until all the CPNs are in the list.

(9) Append all the OBNs to the sequence in a decreasing order of *b-level*.

Using the CPN-Dominant List, we can schedule the nodes on the list one after another to the processors. Again, in order not to incur high complexity, we do not search for the earliest slot on a processor but simply schedule a node to the ready-time of a processor. Initially, the ready-time of all available processors is zero. After a node is scheduled to a processor, the ready-time of that processor is updated to the finish-time of the last node. By doing so, a node is scheduled to a processor that allows the earliest start-time, which is determined by checking the processor's ready-time with the node's data arrival time (DAT). The DAT of a node can be computed by taking the maximum value among the message arrival times across the parent nodes. If the parent is scheduled to the same processor as the node, the message arrival time is simply the parent's finish-time; otherwise it is equal to the parent's finish-time (on a remote processor) plus the communication cost of the edge. Not all processors need to be checked in this process. Instead, we can examine the processors accommodating the parent nodes together with an empty processor (if any). The procedure for generating the initial schedule can be formalized below.

InitialSchedule:

(1) Construct the CPN-Dominant List;

Repeat

(2) Remove the first node n_i from the list;

(3) Schedule n_i to the processor, among the processors accommodating the parent nodes of n_i together with a new processor (if any), that allows the earliest start-time by checking n_i's DAT with the ready-times of the processors;

Until the list is empty;

The time-complexity of *InitialSchedule* is derived as follows. The first step takes $O(e+v)$ time. In the repeat loop, the dominant step is the procedure to determine the data arrival time of a node. The cumulative time-complexity of this step throughout the execution of the repeat loop is also $O(e+v)$ because each edge is visited once. Thus, the overall time-complexity of *InitialSchedule* is also $O(e+v)$.

Given the procedure *InitialSchedule* we present the sequential version of our neighborhood search algorithm. In order to avoid the algorithm being trapped in a local optimal solution, we incorporate a probabilistic jump procedure in the algorithm. The FAST algorithm is outlined below.

The FAST Algorithm:

(1) *NewSchedule = InitialSchedule*

(2) Construct the blocking-nodes list which contains all the IBNs and OBNs;

(3) BestSL = infinity; searchcount = 0;

(4) repeat

(5) *searchstep* = 0; *counter* = 0;

(6) **do** { /* neighborhood search */

(7) Pick a node n_i randomly from the blocking-nodes list;

(8) Pick a processor *P* randomly;

(9) Transfer n_i to *P*;

(10) If schedule length does not improve, transfer n_i back to its original processor and increment *counter*; otherwise, set *counter* to 0;

(11) } **while** (*searchstep*++ < MAXSTEP and *counter* < MARGIN);

(12) **if** BestSL > SL(NewSchedule) **then**

(13) *BestSchedule = NewSchedule*

(14) BestSL = SL(NewSchedule)

(15) **endif**

(16) NewSchedule = Randomly pick a node from the CP and transfer it to another processor; /* probabilistic jump */

(17) **until** (searchcount++ > MAXCOUNT);

The total number of search-steps is $MAXSTEP \times MAXCOUNT$. While the number of search steps in each iteration is bounded by MAXSTEP, the algorithm will also terminate searching and proceed to the step of probabilistic jump if the solution does not improve within a specified number of steps, denoted as MARGIN. This is done in order to further enhance the expected

†. The *b-level* of a node is the length (sum of the computation and communication costs) of the longest path from this node to an exit node. The *t-level* of a node is the length of the longest path from an entry node to this node (excluding the cost of this node)

efficiency of the algorithm.

The reason of making MAXSTEP, MARGIN, and MAXCOUNT as constants is two-fold. First, the prime objective in the design of the algorithm is to keep the time-complexity low even when the size of the input graph is huge. Second, the major strength of the FAST algorithm lies in its ability to generate a good initial solution by using the CPN-Dominant List. As such, the likelihood of improving the initial solution dramatically by using large number of search steps is not high. Thus, we fix MARGIN to be 2, MAXSTEP to be 8, and MAXCOUNT to be 64.

The time-complexity of the sequential FAST algorithm is determined as follow. As discussed earlier, the procedure *InitialSchedule*() takes $O(e+v)$ time. The blocking-nodes list can be constructed in $O(v)$ time as the IBNs and OBNs are already identified in the procedure *InitialSchedule*(). In the main loop, the node transferring step takes $O(e+v)$ time since we have to re-visit all the edges once after transferring the node to a processor in the worst case. Thus, the overall time-complexity of the sequential algorithm is $O(e+v)$.

3.3 Parallel Probabilistic Search

The parallelization of the neighborhood search is based on a partitioning of the blocking-nodes set into q subsets, where q is the number of available *physical processing elements* (PPEs), on which the PFAST algorithm is executed. Each PPE then performs a neighborhood search using its own blocking-nodes subset. The PPEs communicate periodically to exchange the best solution found thus far and start new search steps based on the best solution. The period of communication for the PPEs is set to be T number of search-steps, which follows an exponentially decreasing sequence: initially $\left\lceil \frac{\tau}{2} \right\rceil$, then $\left\lceil \frac{\tau}{4} \right\rceil$, $\left\lceil \frac{\tau}{8} \right\rceil$, and so on, where $\tau = \left\lceil \frac{MAXCOUNT}{q} \right\rceil$. The rationale is that at early stages of the search, exploration is more important than exploitation. The PPEs should, therefore, work independently for longer period of time. However, at final stages of the search, exploitation is more important so that the PPEs should communicate more frequently.

The PFAST algorithm is outlined below.

The PFAST Algorithm:

```
(1)  if myPPE() == master then
(2)      Determine the initial schedule;
(3)      Construct the blocking-nodes set;
(4)      Partition the blocking-nodes set into q
         subsets which are ordered topologically;
(5)  endif
(6)  Every PPE receives a blocking-nodes subset
     and the initial schedule;
(7)  repeat
(8)      i = 2
(9)      repeat /* search */
(10)         Run FAST to search for a better
             schedule;
(11)     until searchcount > ⌈MAXCOUNT / (i × q)⌉;
(12)     Exchange the best solution;
(13) until total searchcount = MAXCOUNT;
```

In the PFAST algorithm, one PPE is designated as the master, which is responsible for preprocessing work including construction of an initial schedule, the blocking-nodes set, and the subsets.

Since the total number of search-steps is evenly distributed to the PPEs, the PFAST algorithm should have linear speedup over the sequential FAST algorithm if communication takes negligible time. However, inter-PPE communication inevitably takes significant amount of time and the ideal case of linear speedup is not achievable. But the solution quality of PFAST can be better than that of the sequential FAST algorithm. This is because the PPEs explore different parts of the search space simultaneously through different neighborhoods induced by the partitions of the blocking-nodes set. The sequential FAST algorithm, on the other hand, has to handle a much larger neighborhood for the same problem size.

4 A Scheduling Example

To see how the procedure *InitialSchedule* works, consider the DAG shown earlier in Figure 1. The attributes used by the other four algorithms are also shown in Figure 2. The CPN-Dominant List of the DAG is $\{n_1, n_3, n_2, n_7, n_6, n_5, n_4, n_8, n_9\}$. Note that n_8 is considered after n_6 because n_6 has a smaller value of *t-level*. Using the CPN-Dominant List, the initial schedule produced by *InitialSchedule* is shown in Figure 3(a). The schedule length generated by *InitialSchedule* is already short, despite its simple scheduling strategy.

node	SL	*t-level*	*b-level*	*ALAP*
$^*n_1$	12	0	37	0
n_2	8	6	23	14
n_3	8	3	23	14
n_4	9	3	20	17
n_5	10	3	30	7
n_6	5	10	15	22
$^*n_7$	5	22	15	22
n_8	5	18	15	22
$^*n_9$	1	36	1	36

Figure 2: The static levels (SLs), *t-levels*, *b-levels*, and ALAP times of the nodes (CPNs are marked by an asterisk).

To illustrate the effectiveness of the neighborhood search process, consider the initial schedule shown in Figure 3(a). The blocking-nodes list of the DAG is $\{n_2, n_3, n_4, n_5, n_6, n_8\}$. We can notice that the node n_6 blocks the CPN n_9. In the random search process it is highly probable that n_6 is selected for transferring. Suppose it is transferred from PE 1 to PE 3. The resulting schedule is shown in Figure 3(b), from which we can see that despite the increased start times of n_5 and n_8, the final schedule length is nonetheless shortened.

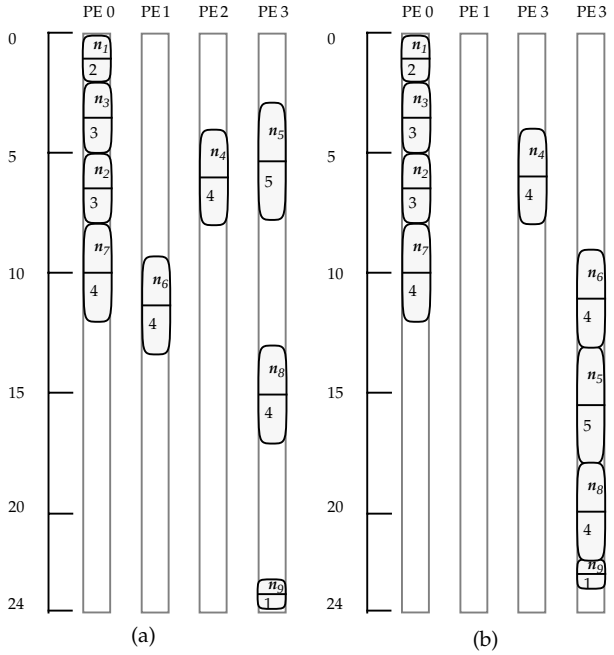


Figure 3: (a) Schedules generated by the *InitialSchedule*() (schedule length = 24); (b) The final schedule after the local search process with node n_6 is transferred to PE 3 (schedule length = 23).

5 Performance Results

In our study, by using a prototype computer-aided scheduling tool called CASCH, we compared our proposed algorithm with four related scheduling algorithms: the Mobility Directed (MD) algorithm [20], the Earliest Task First (ETF) algorithm [9], the Dynamic Level Scheduling (DLS) algorithm [19], and the Dominant Sequence Clustering (DSC) algorithm [21]. We chose the DSC, MD, ETF, and DLS algorithms out of 14 algorithms which we compared in a previous study [3]. The comparison of FAST with these algorithms provides an indirect comparison with the remaining 10 algorithms.

In this section we first present the performance results of the sequential FAST algorithm and compare it with those of DSC, MD, ETF, and DLS algorithms using a prototype parallelization tool. In Section 5.3 we present the performance results of the PFAST algorithm by using two suites of random task graphs for which optimal solutions are known. In Section 5.4 we present the results of applying the algorithm to large DAGs. For comparison, the results of the DLS, DSC, and ETF algorithms are also shown.

5.1 CASCH

We performed experiments using the CASCH tool, which generates a task graph from a sequential program, uses a scheduling algorithm to perform scheduling, and then generates the parallel code in a scheduled form for the Intel Paragon. The timings for the nodes and edges on the DAG are assigned through a timing database which was obtained through profiling. CASCH also provides a graphical interface to interactively run and test various algorithms including the ones discussed in this paper. Instead of only measuring the schedule length through a Gantt chart, we measure the running time of the scheduled code on the Paragon. Various scheduling algorithms, therefore, can be more accurately tested and compared through CASCH using real applications on an actual machine. The reader is referred to [2] for details about the tool.

In addition, in order to examine the performance of the algorithm given very large graphs which can arise in practice, we performed experiments with randomly generated large DAGs consisting of thousands of nodes.

5.2 Parallel Applications

In our first experiment we tested the FAST algorithm with the DAGs generated from three real applications: Gaussian elimination, Laplace equation solver and Fast Fourier Transform (FFT) [3]. The Gaussian elimination and Laplace equation solver applications operate on matrices. Thus, the number of nodes in the DAGs generated from these applications are related to the matrix dimension *N* and is about $O(N^2)$. On the other hand, the FFT application accepts the number of points as input. We examined the performance in three aspects: application execution time, number of processors used and the scheduling algorithm running time.

The results for the Gaussian elimination are shown in Figure 4. In Figure 4(a), we normalized the application execution times obtained through all the algorithms with respect to those obtained through the FAST algorithm. It was shown that the programs scheduled by the FAST algorithm are 3% to 15% faster than the other algorithms. Note that the results of the DSC algorithm for matrix dimensions 16 and 32 were not available because the DSC used more than the available Paragon processors in

Algorithm	Matrix Dimension 4	8	16	32
FAST	1.00	1.00	1.00	1.00
DSC	1.05	1.08	N.A.	N.A.
MD	1.00	1.03	1.08	1.10
ETF	1.00	1.07	1.10	1.15
DLS	1.00	1.08	1.10	1.14

(a) Normalized execution times of Gaussian elimination on the Intel Paragon.

Algorithm	Matrix Dimension 4	8	16	32
FAST	4	8	16	32
DSC	5	22	95	128
MD	2	3	4	7
ETF	3	7	16	32
DLS	3	7	16	32

(b) Number of Processors used for the Gaussian elimination.

Algorithm	Matrix Dimension (Number of Tasks) 4 (20)	8 (54)	16 (170)	32 (594)
FAST	0.06	0.09	0.15	0.52
DSC	0.04	0.06	0.09	0.21
MD	6.33	6.85	39.54	266.89
ETF	0.02	0.06	0.24	2.41
DLS	0.08	0.09	0.42	4.00

(c) Scheduling times (sec) on a SPARC Station 2 for the Gaussian elimination.

Figure 4: Normalized execution times, number of processors used, and scheduling algorithm running times for the Gaussian elimination for all the scheduling algorithms.

scheduling the parallel program. This can be explicated by the fact that the DSC algorithm uses $O(v)$ processors. Concerning the number of processors used, the FAST, ETF and DLS algorithms used about the same amount of processors. The number of processors used by all the algorithms is shown in Figure 4(b). The scheduling times of all the algorithms are shown in Figure 4(c) indicating that the DSC algorithm was the fastest algorithm with the proposed FAST algorithm very close to it. On the other hand, the ETF and DLS algorithms running times are relatively large but they were much faster than the MD algorithm. This is because the MD algorithm is about $O(v)$ times slower than the other algorithms.

The results for the Laplace equation solver are shown in Figure 4, from which we can see that the percentage improvements of the FAST algorithm over the other algorithms is up to 25%. As to the number of processors used, the FAST, MD, ETF and DLS algorithms demonstrated similar performance with the DSC algorithm again uses more processors than the other algorithms. For the scheduling times, the FAST algorithm is the fastest among all the algorithms. The MD algorithm is again $O(v)$ times slower than the other algorithms.

Algorithm	Matrix Dimension 4	8	16	32
FAST	1.00	1.00	1.00	1.00
DSC	1.00	1.09	1.13	1.21
MD	1.00	1.12	1.15	1.25
ETF	1.00	1.11	1.14	1.24
DLS	1.00	1.10	1.13	1.23

(a) Normalized execution times of Laplace equation solver on the Intel Paragon.

Algorithm	Matrix Dimension 4	8	16	32
FAST	1	4	7	14
DSC	1	13	37	64
MD	1	5	8	13
ETF	1	5	8	16
DLS	1	5	8	15

(b) Number of Processors used for the Laplace equation solver.

Algorithm	Matrix Dimension (Number of Tasks) 4 (18)	8 (66)	16 (258)	32 (1026)
FAST	0.05	0.09	0.35	1.28
DSC	0.07	0.11	0.40	4.29
MD	6.23	7.64	111.46	768.90
ETF	0.04	0.05	0.28	3.06
DLS	0.06	0.11	0.55	5.33

(c) Scheduling times (sec) on a SPARC Station 2 for the Laplace equation solver.

Figure 5: Normalized execution times, number of processors used, and scheduling algorithm running times for the Laplace equation solver for all the scheduling algorithms.

The results for the FFT are shown in Figure 4. The FAST algorithm is again better than all the other four algorithms in terms of the application execution times and scheduling times.

5.3 Comparison against Optimal Solutions

In this section, we present the performance results of the PFAST algorithm. We implemented the PFAST algorithm on the Intel Paragon using the C language and tested it using different suites of synthetic task graphs. Our aim is to investigate the absolute solution quality of the algorithm by applying it to two different sets of random

Algorithm	Number of Points 16	64	128	512
FAST	1.00	1.00	1.00	1.00
DSC	1.03	1.08	1.10	1.15
MD	1.04	1.09	1.11	1.17
ETF	1.02	1.08	1.10	1.15
DLS	1.03	1.07	1.09	1.14

(a) Normalized execution times of FFT on the Intel Paragon.

Algorithm	Number of Points 16	64	128	512
FAST	5	12	9	23
DSC	5	12	13	25
MD	5	10	6	21
ETF	3	10	11	11
DLS	7	10	11	11

(b) Number of Processors used for the FFT.

Algorithm	Number of Points (Number of Tasks) 16 (14)	64 (34)	128 (82)	512 (194)
FAST	0.06	0.10	0.12	0.19
DSC	0.07	0.08	0.07	0.10
MD	6.38	9.09	9.87	75.17
ETF	0.05	0.08	0.09	0.16
DLS	0.05	0.18	0.20	0.67

(c) Scheduling times (sec) on a SPARC Station 2 for FFT.

Figure 6: Normalized execution times, number of processors used, and scheduling algorithm running times for FFT for all the scheduling algorithms.

task graphs for which optimal solutions are known. As no widely accepted benchmark graphs exist for the DAG scheduling problem, we believe using random graphs with diverse parameters is appropriate for testing the performance of the algorithm.

The first suite of random task graphs consists of three sets of graphs with different CCRs: 0.1, 1.0, and 10.0. Each set consists of graphs in which the number of nodes vary from 10 to 32 with increments of 2, thus, totalling 12 graphs per set. The graphs within the same set have the same value of CCR. The graphs were randomly generated as follows: First the computation cost of each node in the graph was randomly selected from a uniform distribution with mean equal to 40 (minimum = 2 and maximum = 78). Beginning with the first node, a random number indicating the number of children was chosen from a uniform distribution with mean equal to $\frac{v}{10}$, thus, the connectivity of the graph increases with the size of the graph. The communication cost of each edge was also randomly selected from a uniform distribution with mean equal to 40 times the specified value of CCR. Hereafter this suite of graphs is designated type-1 random task graphs.

To obtain optimal solutions for the task graphs, we applied a parallel A* algorithm [12] to the graphs. Since generating optimal solutions for arbitrarily structured task graphs takes exponential time, it is not feasible to obtain optimal solutions for large graphs. On the other hand, to investigate the scalability of the PGS algorithm, it is desirable to test it with larger task graphs for which optimal solutions are known. To resolve this problem, we employed a different strategy to generate the second suite of random task graphs. Rather than trying to find out the optimal solutions after the graphs are randomly generated, we set out to generate task graphs with *given* optimal schedule lengths and number of processors used in the optimal schedules.

The method of generating task graphs with known optimal schedules is as follows: Suppose that the optimal schedule length of a graph and the number of processors used are specified as SL_{opt} and p, respectively. For each PE i, we randomly generate a number x_i from a uniform distribution with mean $\frac{v}{p}$. The time interval between 0 and SL_{opt} of PE i is then randomly partitioned into x_i sections. Each section represents the execution span of one task, thus, x_i tasks are "scheduled" to PE i with no idle time slot. In this manner, v tasks are generated so that every processor has the same schedule length. To generate an edge, two tasks n_a and n_b are randomly chosen such that $FT(n_a) < ST(n_b)$. The edge is made to emerge from n_a to n_b. As to the edge weight, there are two cases to consider: (i) the two tasks are scheduled to different processors, and (ii) the two tasks are scheduled to the same processor. In the first case the edge weight is randomly chosen from a uniform distribution with maximum equal to $(ST(n_b) - FT(n_a))$ (the mean is adjusted according to the given CCR value). In the second case the edge weight can be an arbitrary positive integer because the edge does not affect the start and finish times of the tasks which are scheduled to the same processor. We randomly chose the edge weight for this case according to the given CCR value. Using this method, we generated three sets of task graphs with three CCRs: 0.1, 1.0, and 10.0. Each set consists of graphs in which the number of nodes vary from 50 to 500 in increments of 50; thus, each set contains 10 graphs. The graphs within the same set have the same value of CCR. Hereafter we call this suite of graphs the type-2 random task graphs.

Table 1 shows the results of the PFAST algorithm using 1, 2, 4, 8, and 16 PPEs on the Intel Paragon. Using 1 PPE means that the algorithm is the sequential FAST

Table 1: Results of the PFAST algorithm compared against optimal solutions (% deviations) for the type-1 random task graphs with three CCRs using 1, 2, 4, 8, and 16 PPEs on the Intel Paragon.

CCR		0.1					1.0					10.0				
No. of PPEs		1	2	4	8	16	1	2	4	8	16	1	2	4	8	16
Graph Size	10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	5.56	0.00	0.00	0.00	0.00	0.00
	12	8.67	8.67	8.67	8.67	11.05	7.29	7.29	7.29	7.29	12.92	0.00	0.00	0.00	0.00	5.05
	14	0.00	0.00	0.00	0.00	8.07	7.76	7.76	7.76	7.76	10.01	0.00	0.00	0.00	0.00	14.12
	16	13.64	13.64	13.64	13.64	13.64	0.00	0.00	0.00	0.00	0.00	29.26	29.26	29.26	29.26	29.26
	18	12.99	12.99	12.99	12.99	12.99	0.00	0.00	0.00	0.00	0.00	18.90	18.90	18.90	18.90	31.00
	20	0.00	0.00	0.00	0.00	0.00	0.33	0.33	0.33	0.33	0.33	0.00	0.00	0.00	0.00	0.00
	22	0.00	0.00	0.00	0.00	0.00	4.31	4.31	4.31	4.31	4.31	7.96	7.96	7.96	7.96	10.95
	24	13.50	13.50	13.50	13.50	13.50	14.06	14.06	14.06	14.06	14.06	21.35	21.35	21.35	21.35	35.06
	26	9.45	9.45	9.45	9.45	15.10	6.50	6.50	6.50	6.50	9.88	16.22	16.22	16.22	16.22	27.68
	28	0.00	0.00	0.00	0.00	0.00	12.58	12.58	12.58	12.58	12.58	23.00	23.00	23.00	23.00	23.00
	30	10.01	10.01	10.01	10.01	15.28	19.86	19.86	19.86	19.86	19.86	13.70	13.70	13.70	13.70	13.70
	32	13.70	13.70	13.70	13.70	17.67	10.44	10.44	10.44	10.44	10.44	29.60	29.60	29.60	29.60	29.60
No. of Opt.		5	5	5	5	4	3	3	3	3	2	4	4	4	4	2
Avg. Dev.		11.71	11.71	11.71	11.71	13.41	9.24	9.24	9.24	9.24	10.00	20.00	20.00	20.00	20.00	21.94

algorithm. It should be noted that for graphs of smaller size, the blocking-nodes subsets of the PPEs are not disjoint so as to make each subset contain at least 2 nodes. In the table, the total number of optimal solutions generated and the average percentage deviations (from the optimal solutions) for each CCR are also shown. Note that the average percentage deviations are calculated by dividing the total deviations by the number of non-optimal cases only. These average deviations thus indicate more accurately the performance of the PFAST algorithm when it is not able to generate optimal solutions. We notice that the PFAST algorithm generated optimal solutions for a significant portion of all the test cases. While the number of optimal solutions is about the same for the three values of CCR, the average deviations increase with the larger values of CCR. The most important observation is that the deviations do not vary much with increasing numbers of PPEs used. For some of the graphs, the deviations do not change for any number of PPEs used. An explanation for this phenomenon is that the final solutions of such cases can be reached within a few transferal of blocking-nodes. Another observation is that when 16 PPEs were used, the deviations of some cases increased. This is presumably due to the small sizes of blocking-nodes subsets which restrict the diversity of the random search. Finally we note that the worst percentage deviation (for the case of CCR=10.0, graph size=16) is within 30% from the optimal. Thus, the PFAST algorithm has shown reasonably high capability in generating near-to-optimal solutions.

The average execution times and speedups of the PFAST algorithm are shown in Figure 7. These averages were computed across the three values of CCR. We notice that the execution times of the PFAST algorithm using 1 PPE is already very short. Furthermore, the speedup curves are quite flat indicating that the speedup of the algorithm is not affected by increasing graph sizes. Another observation is that the speedups are smaller than the ideal linear speedups by a considerable margin. An explanation for these two observations is that the sizes of the type-1 random graphs are small so that the amount of scheduling effort does not dominate the amount of inter-PPE communication time. In other words, the inter-PPE communication is a significant overhead that limits the achievable speedups. However, even for such small graphs, the PFAST algorithm exhibited reasonable scalability.

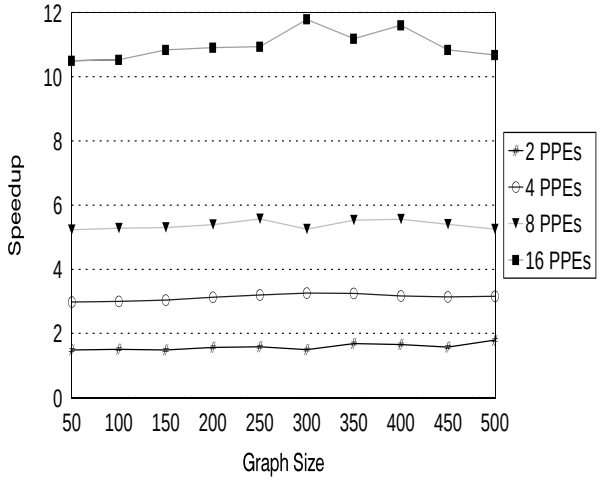
Table 2 shows the results of the PFAST algorithm for the type-2 random task graphs using 1, 2, 4, 8, and 16 PPEs on the Intel Paragon. For these much larger graphs, the PFAST algorithm generated only one optimal solution (the case of CCR = 0.1, size = 50). However, an encouraging observation is that the percentage deviations are small. Indeed, the best deviation is only 2.43% and the worst only 45.01%, which can be considered as close-to-optimal. The average deviations, which increase with increasing CCR, are smaller than 30%. A very interesting observation is that for some cases of larger task graphs (e.g., sizes larger than 200), using more PPEs improve the schedule lengths. For example, in the case of CCR = 0.1 and size = 300,

Table 2: Results of the PFAST algorithm compared against optimal solutions (% deviations) for the type-2 random task graphs with three CCRs using 1, 2, 4, 8, and 16 PPEs on the Intel Paragon.

CCR		0.1					1.0					10.0				
No. of PPEs		1	2	4	8	16	1	2	4	8	16	1	2	4	8	16
Graph Size	50	0.00	0.00	0.00	0.00	0.00	5.34	5.34	5.34	5.34	5.34	29.26	29.26	29.26	29.26	29.26
	100	5.57	5.57	5.57	5.57	5.57	14.90	14.90	14.90	14.90	14.90	45.01	45.01	45.01	45.01	45.01
	150	17.80	17.80	17.80	17.80	17.80	16.83	16.83	16.83	16.83	16.83	17.13	17.13	17.13	17.13	17.13
	200	13.69	13.69	13.69	13.69	13.69	19.02	19.02	19.02	18.08	16.07	20.34	20.34	20.34	15.28	13.57
	250	2.83	2.83	2.83	2.83	2.43	26.97	26.97	26.97	25.99	22.11	34.06	34.06	34.06	25.67	29.43
	300	18.79	18.79	18.79	16.56	14.27	25.30	25.30	25.30	23.28	20.12	22.93	22.93	22.93	15.84	14.45
	350	17.20	17.20	17.20	15.12	14.78	25.11	25.11	25.11	25.01	24.62	38.94	38.94	38.94	33.86	22.53
	400	16.20	16.20	16.20	14.43	14.20	15.03	15.03	15.03	13.37	11.64	26.58	26.58	26.58	18.79	14.31
	450	7.04	7.04	7.04	6.22	5.46	26.38	26.38	26.38	25.53	26.38	33.95	33.95	33.95	24.65	19.46
	500	16.33	16.33	16.33	13.92	14.18	30.71	30.71	30.71	29.31	25.98	35.97	35.97	35.97	32.66	31.65
No. of Opt.		1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
Avg. Dev.		10.50	10.50	10.50	9.65	9.22	17.13	17.13	17.13	16.47	14.98	25.35	25.35	25.35	21.51	19.23

Graph Size	Running Times (secs)
10	0.06
12	0.06
14	0.06
16	0.06
18	0.07
20	0.07
22	0.07
24	0.07
26	0.08
28	0.08
30	0.09
32	0.09

(a) Average running times using 1 PPE.



(b) Average speedups.

Figure 7: (a) The average running times of the PFAST algorithm for the type-1 random task graphs with three CCRs using 1 PPE on the Intel Paragon; (b) the average speedups of the PFAST algorithm for 2, 4, 8, and 16 PPEs.

using 4 PPEs resulted in a deviation of 18.79% while using 8 PPEs gave a deviation of 16.56%. Using 16 PPEs further decreased the deviation to 14.27%. This observation implies that parallelization of a search algorithm can potentially improve the solution quality. This is due to the partitioning of the search neighborhood which lets the search to explore different regions of the search space simultaneously, thereby increasing the likelihood of getting better solutions. There are a few cases in which using more PPEs resulted in an increased deviations, however. For example, for the case of CCR = 10.0 and size = 300, using 4 PPEs gave a deviation of 34.06%, while using 8 PPEs improved the deviation to 25.67%. However, when 16 PPEs were used, the deviation was only 29.43%, which is worse than that of using 8 PPEs but better than that of using 4 PPEs.

The average execution times and speedups of the PFAST algorithm for the type-2 random task graphs are shown in Figure 8. Compared with the speedup plots shown earlier in Figure 7, the speedups for type-2 task graphs are considerably higher. This is due to the fact that inter-PPE communication for larger task graphs is not a significant overhead. Again the PFAST demonstrated reasonably good scalability even for task graphs with 500 nodes.

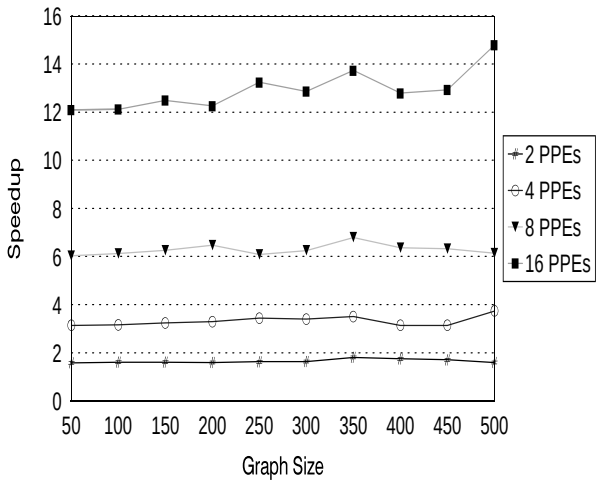
Based on the above results we can conclude that the PFAST algorithm is suitable for finding high quality schedules for large task graphs under strict running times requirements.

5.4 Large DAGs

To test the scalability and robustness of the FAST

Graph Size	Running Times (secs)
50	0.12
100	0.49
150	0.98
200	1.83
250	2.98
300	4.43
350	5.98
400	7.12
450	9.15
500	11.23

(a) Average running times using 1 PPE.



(b) Average speedups.

Figure 8: (a) The average running times of the PGS algorithm for the type-2 random task graphs with three CCRs using 1 PPE on the Intel Paragon; (b) the average speedups of the PGS algorithm for 2, 4, 8, and 16 PPEs.

algorithm we performed experiments with very large DAGs. These DAGs include a 10728-node Gaussian elimination graph, a 10000-node Laplace equation solver graph, a 12287-node FFT graph, and a 10000-node random graph. For these graphs we simply measured the schedule length produced by an algorithm. We applied the DLS, DSC, ETF, and PFAST algorithms to these graphs on the Intel Paragon. We ran the PFAST algorithm using 16 PPEs and other algorithms 1 PPE.

The schedule lengths for the large DAGs, normalized with respect to that of the PFAST algorithm, are shown in Figure 9(a). Note that the MD algorithm was excluded from the comparison because it took more than 8 hours to produce a schedule for a 2000-node DAG. An encouraging observation is that the PFAST algorithm outperformed all the algorithms in all the test cases. The percentage improvement ranges from 8% to 23%. Concerning the scheduling times, we can immediately note from Figure 9(b) that the ETF and DLS algorithms were considerably slower than the PFAST and DSC algorithms. By using effective parallelization, the PFAST algorithm outperforms the DSC algorithm both in terms of solution quality and complexity. These results of large DAGs indeed provide further evidence to the claim that the PFAST algorithm is suitable for finding high quality schedules for huge DAGs.

	Graph types (Number of Nodes)			
Algorithm	Gauss (10728)	Laplace (10000)	FFT (12287)	Random (10000)
PFAST	1.00	1.00	1.00	1.00
DSC	1.12	1.23	1.21	1.15
ETF	1.08	1.20	1.18	1.12
DLS	1.07	1.20	1.18	1.10

(a) Normalized schedule lengths for large DAGs; the PFAST algorithm used 16 PPEs on the Intel Paragon.

	Graph types (Number of Nodes)			
Algorithm	Gauss (10728)	Laplace (10000)	FFT (12287)	Random (10000)
PFAST	30.24	31.68	48.88	40.68
DSC	298.34	228.23	600.23	463.42
ETF	6059.69	8235.23	10234.21	9324.82
DLS	16377.28	22877.40	29877.35	21908.43

(b) Scheduling times (sec) on the Intel Paragon; the PFAST algorithm used 16 PPEs while other algorithms used 1 PPE.

Figure 9: Normalized schedule lengths and scheduling times for the large DAGs for all the scheduling algorithms.

6 Concluding Remarks

In this paper we have presented a low complexity parallel algorithm, called the PFAST algorithm, to meet the conflicting goals of high performance and low time-complexity. Instead of using sophisticated methods to optimize the scheduling of individual nodes, the PFAST algorithm first generates an initial schedule and then refines it in parallel using an effective probabilistic search techniques.

We have compared the algorithm with a number of well-known efficient scheduling algorithms using real applications and randomly generated task graphs. The results obtained demonstrate that the proposed algorithm is superior to existing algorithms in terms of both solution quality and complexity. Based on the comparison study in this paper and the comparison of 14 algorithms in [3], we find the PFAST algorithm to be the fastest algorithm known to us.

An interesting observation of the PFAST algorithm is that parallelization can sometimes improve solution quality in that for some cases the PFAST algorithm generated better solutions when using more PPEs. This is

due to the partitioning of the blocking-nodes set, which implies a partitioning of the search neighborhood. The partitioning causes the algorithm to explore the search space simultaneously, thereby enhancing the likelihood of getting better solutions.

References

[1] T.L. Adam, K.M. Chandy, and J. Dickson, "A Comparison of List Scheduling for Parallel Processing Systems," *Communications of the ACM*, vol. 17, Dec. 1974, pp. 685-690.

[2] I. Ahmad, Y.-K. Kwok, and M.-Y. Wu, "Analysis, Evaluation, and Comparison of Algorithms for Scheduling Task Graphs on Parallel Processors," *International Symposium on Parallel Architectures, Algorithms, and Networks*, Beijing, China, Jun. 1996, pp. 207-213.

[3] I. Ahmad, Y.-K. Kwok, M.-Y. Wu, and W. Shu, "Automatic Parallelization and Scheduling of Programs on Multiprocessors using CASCH," Proc. *Int'l Conference on Parallel Processing*, Aug. 1997, to appear.

[4] M. Beck, K.K. Pingali, and A. Nicolau, "Static scheduling for dynamic dataflow machines," *Journal of Parallel and Distributed Computing*, 10, 1990, pp. 279-288.

[5] E.G. Coffman, *Computer and Job-Shop Scheduling Theory*, Wiley, New York, 1976.

[6] H. El-Rewini, H.H. Ali, and T.G. Lewis, "Task Scheduling in Multiprocessing Systems," *IEEE Computer*, Dec. 1995, pp. 27-37.

[7] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, 1979.

[8] A. Gerasoulis and T. Yang, "A Comparison of Clustering Heuristics for Scheduling DAG's on Multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 16, no. 4, Dec. 1992, pp. 276-291.

[9] J.J. Hwang, Y.C. Chow, F.D. Anger, and C.Y. Lee, "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times," *SIAM Journal on Computing*, vol. 18, no. 2, Apr. 1989, pp. 244-257.

[10] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Trans. on Computers*, vol. C-33, Nov. 1984, pp. 1023-1029.

[11] Y.-K. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs onto Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 5, May 1996, pp. 506-521.

[12] Y.-K. Kwok, I. Ahmad, M. Kafil, and Imtiaz Ahmad, "Parallel Algorithms for Optimal Scheduling of Arbitrary Task Graphs to Multiprocessors," *submitted for publication*.

[13] B. Lee, A.R. Hurson, and T.Y. Feng, "A Vertically Layered Allocation Scheme for Data Flow Systems," *Journal of Parallel and Distributed Computing*, vol. 11, 1991, pp. 175-187.

[14] J.-C. Liou and M.A. Palis, "A Comparison of General Approaches to Multiprocessor Scheduling," Proc. of 11th *Int'l Parallel Processing Symposium*, Apr. 1997, pp. 152-156.

[15] C. McCreary, A.A. Khan, J.J. Thompson, and M.E. McArdle, "A Comparison of Heuristics for Scheduling DAG's on Multiprocessors," Proceedings of *International Parallel Processing Symposium*, 1994, pp. 446-451.

[16] M.A. Palis, J.-C. Liou, and D.S.L. Wei, "Task Clustering and Scheduling for Distributed Memory Parallel Architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 1, Jan. 1996, pp. 46-55.

[17] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[18] B. Shirazi, M. Wang, and G. Pathak, "Analysis and Evaluation of Heuristic Methods for Static Scheduling," *Journal of Parallel and Distributed Computing*, no. 10, 1990, pp. 222-232.

[19] G.C. Sih and E.A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, Feb. 1993, pp. 75-87.

[20] M.Y. Wu and D.D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems," *IEEE Trans. on Parallel and Distributed Sys.*, vol. 1, no. 3, Jul. 1990, pp. 330-343.

[21] T. Yang and A. Gerasoulis, "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 9, Sep. 1994, pp. 951-967.

[22] J. Yang, L. Bic, and A. Nicolau, "A Mapping strategy for MIMD Computers," *International Journal of High Speed Computing*, vol. 5, no. 1, 1993, pp. 89-103.